CONLAN—A formal construction method for hardware description languages: language derivation

by ROBERT PILOTY
Technische Hochschule Darmstadt
FR Germany

MARIO BARBACCI
Carnegie-Mellon University
Pittsburgh, Pennsylvania

DOMINIQUE BORRIONE
Universite de Grenoble
Grenoble, France

DONALD DIETMEYER
University of Wisconsin-Madison
Madison, Wisconsin

FREDRICK HILL
University of Arizona
Tucson, Arizona

and

PATRICK SKELLY
Honeywell
Phoenix, Arizona

INTRODUCTION

A CONLAN document has significance only if it is read by a person or machine. That reader (environment) is required to use available facilities to respond to and interact with the document. It must provide the type checking mechanism. It must record the names of defined and declared items and provide the data base they require. It must record signal values. From such records, it can determine facts of importance to continued document evaluation. "System interfaces" are prescribed environment responses, not formally defined via CONLAN syntax.

1.1 Values, signals, and carriers

Three broad classes of objects are of primary concern in working members of the CONLAN family:

"Values" are static objects; they do not change with time. An integer, a character, etc. are values.

"Signals" are lists of values. A different time is associated with each value. A signal is then a history of values.

"Carriers" are containers for values or signals. These values or signals can be replaced as a result of an operation invocation.

1.2 CONLAN model of time

CONLAN provides a discrete model of continuous, real time.

Real time is broken into uniform durations called "intervals" identified with integers greater than zero. Ascending, successive integers are associated with contiguous intervals. No relation between the interval and the real time second exists in general. An implementation may impose such a relation or permit users to specify such a relation.

At the beginning of each interval there are an indefinite number of computation "steps" identified with integers greater than zero. Successive steps provide a before/after relation only.

Values obtained at the last step of computation are the values associated with the interval.

When modeling a specific digital system, satisfactory results are obtained at reasonable computational cost by quantizing time to some fraction of the second; for purposes of example assume the nanosecond. Actual signals are then constrained by this quantization to change at the boundaries

of 1 ns. durations. Computing the value of a specific signal during a specific 1 ns. duration may require successive computations: if a wire is driven by a gate network modelled by $c\wedge b \vee c\wedge d$, then $a\wedge b$ and $c\wedge d$ must be evaluated before the signal value is determined. The CONLAN interval and step support this model of digital hardware and method of simulation (Figure 1).

No real time is thought to elapse when evaluating a mathematical function or executing a computer program. Yet many successive computational steps are usually required. Again the CONLAN model of time supports such computation.

2. FORMAL DERIVATION OF SIGNALS

In order to model real hardware components, some mechanism to describe delays in components and wires must be provided. The solution adopted in CONLAN is to keep the history of values computed at every step of every interval. Separate histories (called 'signals' in CONLAN) are kept for each component, pin, wire, etc. of the hardware system. Signals are abstractions and do not have a physical interpretation. To provide the link between the signal (i.e. a history of values) and the component, a special type of object, called a 'signal_carrier' is provided by the language. In this chapter we formally define signals as a bcl type together with operations to manipulate signals. Signal _carriers (carriers, for short) are the subject of the following chapter.

2.1 CONLAN model of computation

Hardware descriptions record how the signal parts of some carriers are related to those of other carriers. These relations display behavior and/or organization and support computation of unknown signal parts. Such computation is usually performed viewing past and present signal values as "known" and future values as "unknown." With each computational step, known values are used to determine a future value and thereby change its status to known.

The interval and step counters are managed by the environment. The contents of these counters are made available to toolmakers via $t@$ and $s@$.

$t@$ is an integer whose value is the current time interval. Contiguous values are provided in ascending order starting with one. $s@$ is an integer whose value is the current computation step. Contiguous values are provided in ascending order, starting with one.

When the environment determines that all signals have attained stable values, it increments the value provided by $t@$ and resets the $s@$ counter to 1. It detects computation step oscillation ($s@$ reaches a predetermined limit) and responds to it with a message and optionally termination of document evaluation or continuation using the signal values available at the last step of computation.

The algorithm used by the environment is the following:

Stage	*Action*
1	For each invoked activity and function of the system description under evaluation, determine via the definition of that invoked operation future step values from known present and past signal values. Advance to stage 2.
2	For all carriers which have not been serviced in stage 1, provide for them the missing step value. The determination of the missing value is the responsibility of both the environment and the toolmaker (see finstep@, below). Advance to stage 3.
3	Examine the record of present and next step values. If one or more signals have differing values and $s@$ is less than a predetermined limit, advance the step counter $s@$ and return to stage 1. If $s@$ equals the predetermined limit, publish an "oscillation" error message and (optionally) continue with stage 4; otherwise continue with stage 4.
4	For each invoked activity and function of the system description under evaluation, determine the initial step value for the next interval. The determination of this step value is the responsibility of both the environment and the toolmaker (see finint@, below). Advance to stage 5.
5	Reset $s@$ to 1, increment $t@$, and return to stage 1.

To support the model of computation, the environment uses special operations, finint@ and finstep@ which are provided by the toolmaker.

Finstep@ is an activity which describes the default signal growth mechanism for a computation step. Finint@ is an activity which describes the default signal growth mechanism for a time interval.

None, one or more functions and activities may be invoked in a step for a specific carrier. If multiple invocations attempt to set a signal to different values, a "collision" exists and will be reported as an error. Operations finstep@ and finint@ are independent of invocations; they provide a means of providing default values or propagating values to future steps when no activities are invoked to do so.

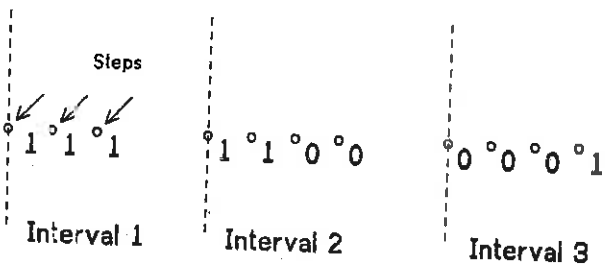


Figure 1—CONLAN model of time.

2.2 Computation step signals

```
TYPE cs_signal@(x: value) BODY                          "/value is the class of all value types/"
                                                        "/definition of the new type elements/"
  ALL a: tytuple@(x) WITH size@(a) > 0 ENDALL

  CARRY =, ≠, size@ ENDCARRY                            "/imported operations from the defining type/"

  FUNCTION select_css(y: cs_signal@(x), s: pint): x     "/access elements (values) in a cs_signal/"
    RETURN old@(y)[s]
    FORMAT@
      EXTEND@ ref_to_declared.5
        ref_to_declared = exp10 :id1 '{' exp7 :id2 '}'
      MEANS@ select_css (id1, id2) ENDFORMAT
  ENDselect_css

  FUNCTION extend_css(y: cs_signal@(x), s: pint, v: x): cs_signal@(x)   "/extend a computation step signal/"
    RETURN IF s = size@(y) + 1 THEN extend(old@(y), v)
      ELIF s ≤ size@(y) THEN IF y{s} ≠ v THEN error@ ELSE y ENDIF       "/collision/"
      ELSE error@ ENDIF                                                 "/order error/"
  ENDextend_css
ENDcs_signal
```

A Computational Step Signal (cs_signal) is the mechanism used to record a history of values during one real time interval. The definition of type cs_signal indicates that the values to be recorded must all be of the same type, and the type must be specified when a cs_signal is declared. Thus, one could have cs_signals recording values of type integer, Boolean, etc.

The type is constructed from a more primitive type (tytuple@, [1]) whose elements are tuples (ordered lists) of elements of the same type. Moreover, cs_signals cannot be empty (size@>0) although they can be of unlimited size. For instance, the set of cs_signals carrying Boolean values is:

```
{
(. 0 .), (. 1 .),                          "/cs_signals of length 1/"
(. 0,0 .), (. 0,1 .), (. 1,0 .), (. 1,1 .)"/cs_signals of length 2/"
(. 0,0,0 .), (. 0,0,1 .), (. 0,1,0 .), ..."/cs_signals of length 3/"
.....
}
```

In addition to carrying a few operations from the defining type ('=', '≠', and size@), cs_signals provide operations for extracting or appending values to a signal.

Function select_css takes two parameters (a cs_signal and a position) and returns the value occupying that position in the signal:

```
RETURN old@(y)[s]
```

The value is extracted using the primitive operation select (defined on elements of tytuple@, with infix notation [..]). This operation however requires that its first parameter be an element of tytuple@ and not an element of cs_signal (or any other type). The type conversion is explicitly done by invoking a primitive operation, old@ which takes an element of a derived type and returns the same element of the defining type.

The format statement describes an extension to the syntax. The extension is expressed in a variation of BNF in which we not only express the syntax but also the semantics of a production. In this case, the modification consists of adding one more alternatives to the definition of the nonterminal 'ref_to_declared'. The new alternative (identified as alternative number 5) indicates that '{' and '}' can be used to invoke the function select_css.

Function extend_css takes three parameters (a cs_signal, a position, and a value). It is used to compute cs_signals based on an existing cs_signal. Arbitrary computations of a cs_signal are not performed. The cs_signal returned may equal the given cs_signal, or be the cs_signal formed from the given cs_signal, extended with the given value on the right.

Extending elements of cs_signal by exactly one position ($s = \text{size@}(y) + 1$) models the computation of step values. Attempts to extend the cs_signal by more than one position violates the model of computation (see error@, below). A reference to a position already in use ($s \leq \text{size@}(y)$) models the simultaneous computation of values from different signal sources. Attempts to change an already recorded value ($y\{s\} \neq v$) however, are reported as collision errors.

Error@ is provided by the environment as part of the system interface. When invoked, the processor of the document issues an error message. Subsequent actions are determined by the sophistication of the environment. All processing might stop; or if the nature of the error is determined, a default value may be returned and processing continued.

For example, assume a cs_signal (a) carrying values of type integer, whose initial history is:

(. 0, −3, 1, 5 .)

Operation	*Result*	
select_css(a,1)	0	Extract the first recorded value.
a{4}	5	Use the syntax extension.
a{5}	error@	The signal contains only four elements.
extend_css(a,1,0)	(. 0,−3,1,5 .)	no change,a{1} was already 0
extend_css(a,1,1)	error@	Collision error.
extend_css(a,5,0)	(. 0,3,1,5,0 .)	Record a new value.

2.3 Real time signals

```
TYPE signals@(x: value) BODY
  tytuple@(cs_signal@(x))
  CARRY =, ≠, size@ ENDCARRY
  FUNCTION select_rts(y: signal@(x), t: pint): cs_signal@(x)
    RETURN old@(y)[t]
    FORMAT@ EXTEND@ ref_to_declared.5 MEANS@ select_rts(id1, id2) ENDFORMAT
  ENDselect_rts

                                                    "/Does a value for interval t, step s exist?/"
  FUNCTION known(y: signal@(x), t, s: pint): bool
    RETURN size@(y) ≥ t ∧ size@(y{t}) ≥ s
  ENDknown

                                                    "/Instantaneous Value/"
  FUNCTION inst_value(y: signal@(x), t, s: pint): x
    ASSERT known(y, t, s) ENDASSERT
    RETURN y{t}{s}
    FORMAT@
      EXTEND@ ref_to_declared.6
      ref_to_declared = exp10 :id1 '{' exp7 :id2 ',' exp7 :id3 '}'
      MEANS@ inst_value(id1, id2, id3)
      EXTEND@ ref_to_declared.7
      ref_to_declared = exp10 : id1 '{' exp7 : id2 ',' '}'
      MEANS@ inst_value(id1,id2,size@(id1{id2}))
      EXTEND@ ref_to_declared.8
      ref_to_declared = exp10 : id1 '{' '}'
      MEANS@ inst_value(id1,size@(id1),size@(id1{size@(id1)}))
    ENDFORMAT
  ENDinst_value

                                                    "/Extend a signal/"
  FUNCTION extend_rts(y: signal@(x), t, s: pint, v: x): signal@(x)
    RETURN IF known(y, t, s) THEN IF y{t, s} ≠ v THEN error@ ELSE y ENDIF
                                                    "/collision/"
      ELIF t = size@(y) THEN THE@ z: signal@(x) WITH@
        size@(z) = size@(y) ∧
        FORALL@ i: bint(1,t−1) IS@ z{i} = y{i}ENDFORALL ∧
        z{t} = extend_css(y{t},s,v; ENDTHE
      ELIF t = size@(y)+1 ∧ s = 1 THEN new@(extend(old@(y), (. v .)) )
      ELSE error@ ENDIF
  ENDextend_rts                                     "/order error/"
  INTERPRETER@ FUNCTION pack@(y: x): signal@(x)
    RETURN THE@ z: signal@(x) WITH@
        size@(z) = t@ ∧
        size@(z{t@}) = s@ ∧
        FORALL@ t: bint(1,t@ − 1) IS@ z{t} = (. y .) ENDFORALL ∧
```

```
        FORALL@ s: bint(1,s@) IS@ z{t@,s} = y ENDFORALL ENDTHE
    ENDpack@
ENDsignal@
```

A Real Time Signal is the mechanism for recording interval values. During an interval, an unlimited number of computation steps may occur and these are recorded in cs_signals. A signal consists of a tuple of these cs_signals.

Signals are derived from type tytuple@. The set of signals is given by the set of all possible tuples whose elements are cs_signals. For instance,

$$\text{signal@(bool)} = \{(.(.0.).),\ldots(.(.0.),(.0.).),\ldots(.(.0,0.),(.0,1.).),\ldots\}$$

Function select_rts takes two parameters (a signal and a position) and returns the element of the signal (a cs_signal) occupying that position. This operation in fact retrieves the entire history of values computed during one interval. The formal statement takes advantage of the extension to the syntax that appeared in the definition of function select_css (in type cs_signal@). In the current extension, no new productions are being added, but rather the semantics of an existing alternative are extended by indicating that '{' and '}' are also used to invoke select_rts. Since the parameter types are different, the type checking mechanism in the language determines which function is to be invoked.

Function known takes three parameters (a signal, an interval number, and a step number) and returns true or false depending on whether there is a value recorded at a given step in a given interval or not. Since cs_signals do not have gaps, all that is needed is to compare the size of tuples with the interval and step numbers.

Function inst_value takes three parameters (a signal, an interval number, and a step number) and returns the value recorded in that step and interval. The ASSERT statement monitors that the function has been properly invoked by asserting that the value is 'known'. If the assertion fails at any time during the invocation of the function, an error is reported by the interpreter. The format statements extend the language by generalizing once again the use of '{' and '}':

Syntax	*Meaning*
css{*s*}	Returns the value recorded during step *s* of a cs_signal.
rts{*i*}	Returns the cs_signal recorded during interval *i* of a signal.
rts{*i*,*s*}	Returns the value recorded during step *s* of interval *i*.
rts{*i*,}	Returns the value recorded during the last step of interval *i*.
rts{ }	Returns the value recorded during the last step of the last interval.

Function extend_rts takes four parameters (a signal, an interval number, a step number, and a value) and computes a signal. As in the case of cs_signals, only restricted computations are performed. If a value has already been recorded at the given step of the given interval, the previous value and the new value must be equal, otherwise a collision is reported by the interpreter.

If the interval referred to is the last interval of the signal, an attempt is made to extend the cs_signal recorded in that position. If the interval referred to is exactly one position beyond the last interval of the signal, and the step number is 1 (i.e. first step) then the signal is extended with a new cs_signal, initialized to the new value. All other cases are excluded and reported as errors. In the expression,

$$\text{extend@(old@(}y\text{),(.}v\text{.))}$$

The second parameter of extend@ illustrates the constant denotation for tuples and derived types (tytuple, cs_signal, signal).

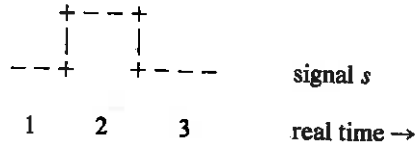
Function pack@ is part of the system interface. By invoking this function, the environment converts a value into a signal. As in all other system interfaces which must be provided by the toolmaker, the definition of pack@ is preceded with the keyword INTERPRETER@. Function pack@ takes one parameter (a value of some type) and returns a signal capable of recording values of the same type. The signal records the history of a value that has remained constant until the current step of the current interval. This function is automatically invoked by the interpreter. It is used to provide automatic type conversion during operation invocations.

For instance, assume a Boolean signal (*s*), with the following history:

$$(.\ (.0,\ 0.),\ (.0,\ 1.),\ (.1,\ 1,\ 0.).)$$

Pictorially, this is represented as:

Real Time Interval	Computation Step	Value
1	1	0
	2	0 last
2	1	0
	2	1 last
3	1	1
	2	1
	3	0 last



Operation	*Result*	
$s\{1\}$	(.0,0.)	
$s\{4\}$	error@	The history of the first interval.
known(s,1,2)	1	A reference to a future interval.
known(s,1,3)	0	There is a value recorded at interval 1, step 2.
$s\{1,2\}$	0	The history of interval 1 does not have three steps.
$s\{1,\}$	0	The value at step 2 of interval 1.
$s\{\}$	0	The value at the last step of interval 1.
extend_rts(s,1,1,0)	s	The value at the last step of the last interval.
extend_rts(s,1,1,1)	error@	The value at step 1 of interval 1 was already 0.
extend_rts(s,3,4,1)	(.(.0,0.),(.0,1.),(.1,1,0,1.).)	Collision, trying to change the history.
extend_rts(s,4,1,1)	(.(.0,0.),(.0,1.),(.1,1,0.),(.1,.).)	Extended version of the current interval.
		Initialize a new interval.

3. FORMAL DERIVATION OF CARRIERS

```
TYPE signal_carrier@(x: value, di: x) BODY
  cell@(signal@(x))
    CARRY put@, empty@ FROM cell@ ENDCARRY
  FUNCTION spart(y: carrier@(x, di)): signal@(x)
    RETURN IF empty@(y) THEN pack@(di) ELSE get@(old@(y)) ENDIF
  ENDspart                                                    "/Signal part/"

  INTERPRETER@ FUNCTION content@(y: carrier@(x, di)): x       "/Present signal value/"
    RETURN IF empty@(y) THEN di ELSE get@(old@(y)){t@, s@}
  ENDcontent

  FUNCTION delay(y: carrier@(x, di), d: pint): x              "/Real time delay/"
    RETURN IF t@ ≤ d THEN di ELSE spart(y){t@-d, } ENDIF
    FORMAT@
      EXTEND@ exp10.10
      exp10 = exp10 :y 'Δ' exp10 :d
      MEANS@ delay(y, d)
    ENDFORMAT
  ENDdelay
ENDsignal_carrier@
```

Type signal_carrier@ (carrier, for short) is derived from primitive type cell@. Each cell contains a signal. Carriers are declared by specifying the type of values recorded by the signal (x) and a default/initial part (di). The role of the default/initial value is to provide a value to be used in some operations, as described below.

Function spart takes one parameter (a carrier) and returns the signal kept in the carrier. Notice the use of the default/initial value to provide a 'signal', even if no history of values has been recorded.

Function content@ is part of the system interface. It takes one parameter (a carrier) and returns the value recorded at step $s@$ of interval $t@$ of the signal (i.e. the 'current' value). It is automatically invoked by the interpreter to provide dereferencing during operation invocations.

Function delay takes two parameters (a carrier and a delay value) and returns the last value of a previous interval. Negative time is avoided by returning the default/initial value when that previous interval would be interval zero or less. The interval number is computed by subtracting the delay parameter from $t@$ (the current interval number). The format statement extends the language by providing an infix notation (Δ) for the delay function.

For example, assume a carrier (x) of Boolean signals with default/initial value 1. Further, assume that the history at $t@=3$, $s@=4$ is the following:

(.(.1,1,1.),(.1,1,0,0.),(.0,0,0,1 .).)

Operation	*Result*	
spart(x)	(.(.1,1,1.),(.1,1,0,0.),(. 0,0,0,1 .).)	
content@(x)	1	Signal
x Δ 5	1	Value at step $s@$, interval $t@$.
x Δ 2	1	Default value.
		Last value of interval 1.

3.1 Terminals

```
TYPE terminal(x: value, def: x) BODY
  carrier@(x, def)
  CARRY content@, delay ENDCARRY
  ACTIVITY connect(y: terminal(x, def), a: x) BODY
    put@(old@(y), extend_rts(spart(old@(y)), t@, s@+1,
    a) )
    FORMAT@
      EXTEND@ activity_invocation.3
      activity_invocation = ref_to_declared :id1 '.='
      expression :id2
      MEANS@ connect(id1, id2)
    ENDFORMAT
  ENDconnect
  INTERPRETER@ ACTIVITY finstep@(y:
  terminal(x,def)) BODY
    IF ¬known(spart(old@(y)), t@, s@+1) THEN y .=
    def ENDIF
  ENDfinstep@
  INTERPRETER@ ACTIVITY finint@(y:
  terminal(x,def)) BODY
    put@(old(y), extend_rts(spart(old@(y)), t@+1, 1,
    content@(y)))
  ENDfinint
ENDterminal
SUBTYPE btm1(default: bool) BODY terminal(bool,
  default) ENDbtm1
SUBTYPE btm0 BODY terminal(bool, 0) ENDbtm0
SUBTYPE btm1 BODY terminal(bool, 1) ENDbtm1
```

Activity connect has two parameters (a terminal and a value). This activity extends the (cs_signal of the current interval of the) signal associated with the terminal.

Activity finstep@ provides a terminal with a default value for the present step of the present interval if none has been provided by an invocation of connect.

Activity finint@ initializes the cs_signal of the next interval of a terminal with the last value of the current interval.

Terminals are carriers with no retention properties. If no connect activity is invoked during a computation step to extend the signal component of a terminal, then the finstep@ activity invoked by the system extends that signal with the default value of the terminal. Activities finstep@ and finint@ force all terminal's signals to grow at the same rate.

Boolean terminals model connection points. Some connection points float high and others float low when not driven. Subtypes btml, btm0, and btm1 are defined as shorthand for commonly used terminals.

3.2 Variables

```
TYPE variable(x: value, init: x) BODY
  carrier@(x, init)
  CARRY content, delay ENDCARRY
  ACTIVITY assign(y: variable(x, init), a: x) BODY
    put@(old@(y),extend_rts(spart(old@(y)), t@, s@+1,
    a))
    FORMAT@
      EXTEND@ activity_invocation.4
      activity_invocation = ref_to_declared :id1 ':='
      expression :id2
      MEANS@ assign(id1, id2)
    ENDFORMAT
  ENDassign
  INTERPRETER@ ACTIVITY finstep@(y:
  variable(x,init)) BODY
    IF ¬known(spart(old@(y)), t@, s@+1) THEN y :=
    content@(y) ENDIF
  ENDfinstep@
  INTERPRETER@ ACTIVITY finint@(y:
  variable(x,init)) BODY
    put@(old@(y), extend_rts(spart(old@(y)), t@+1, 1,
    content@(y)))
  ENDfinit@
ENDvariable
```

Variables have retention properties. They differ from terminals in the role played by function finstep@. If no assign activity is invoked during a computation step to extend the signal component of a variable, then the finstep@ activity invoked by the system extends that signal with the present value. Variables are then much as found in programming languages. Boolean variables may be thought to model abstract storage devices whose value may change at every computation step.

3.3 Real time variables

```
TYPE rtvariable(x: value, init: x) BODY
  carrier@(x, init)
  CARRY content, delay ENDCARRY
  ACTIVITY transfer(y: rtvariable(x, init), a: x) BODY
    put@(old@(y), extend_rts(spart(old@(y)), t@+1, 1,
    a))
    FORMAT@
      EXTEND@ activity_invocation.5
        activity.invocation = ref_to_declared :id1 '<-'
        expression :id2
      MEANS@ transfer(id1, id2)
  ENDFORMAT
  ENDtransfer
  INTERPRETER@ ACTIVITY finstep@(y:
  rtvariable(x,init)) BODY
    put@(old@(y),extend_rts(spart(old@(y)),t@s@+1,
    content@(y)))
  ENDfinstep@
  INTERPRETER@ ACTIVITY finint@(y:
  rtvariable(x,init)) BODY
    IF ¬known(spart(old@(y)), t@+1, 1) THEN y <-
    content@(y) ENDIF
  ENDfinint@
ENDrtvariable
```

Real time variables model abstract storage devices whose value may change only once per real time interval. When

the transfer activity is invoked, the signal part of the real time variable is extended by one real time interval. Within an interval the computation step signal is extended with the first value—all step values are the same. A value is carried from interval to interval when no transfer is invoked.

4. OTHER TYPES IN BASE CONLAN

Pscl makes available for further use four scalar types (int, bool, string, cell@) and tuple@ as the basic structured type [1]. Using the same extension mechanisms presented in this paper, a constructor for arrays has been formally defined and will appear in a forthcoming paper. Its development follows the same pattern used in the development of signals, carriers, terminals, and variables: layers of abstraction are built by defining types, operations, and syntax extensions leading toward the final product. In contrast with the previous developments however, the concept of time steps and intervals does not play a significant role as space, rather than time is being modeled.

Due to space limitations, the full development cannot be given here. Only a sketch of the constituent types will be indicated in Figure 2.

Type tuple@ consists of ordered lists of elements of any type. Type tytuple@(*t*:any@) [1], consists of tuples of elements of the same type, *t*. A particular member of this parameterized type family is type inttuple@ (defined as tytuple@(int)). A range is defined as an inttuple of consecutive, ascending or descending integers.

An array_dimension is a tytuple of ranges. The size of the array dimension is not limited: CONLAN supports arrays of any number of dimensions. An element_index is an inttuple. It describes, with respect to a particular array_dimension the elements along each dimension needed to access a single element of an array. A slice_index is a tytuple@(range). It describes, with respect to a particular array_dimension, the ranges of elements along each dimension needed to access a slice of an array.

Type array(*d*: array_dimension@, *t*:any@) is defined as a tuple of two elements. The dimensions part (of type array_dimension@) describes the dimensions of the array; the value part (of type tytuple@(*t*)) is the list of array elements.

An abbreviated list of operations defined on arrays is depicted below:

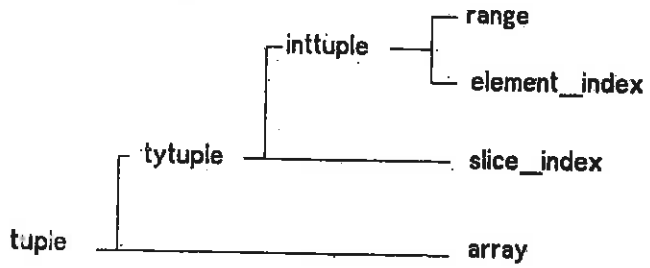


Figure 2—Development of arrays.

select_element(*x*:array(*d*,*t*), *z*: element_index): *t*
  selects an element of an array.
select_slice(*x*:array(*d*,*t*), *z*: slice_index): *t*
  selects a slice of an array.
compatible(*x*, *y*: array(*d*,*t*)): bool
  tests if two arrays have compatible dimensions
equal(*x*, *y*: array(*d*,*t*)): bool
  tests if two arrays have identical value parts.
eq(*x*, *y*: array(*d*,*t*)):array(*d*,bool)
  tests if corresponding elements of compatible arrays are equal.

For many of these operations and types, special infix formats and constant denotation formats have been defined via the syntax extension mechanism.

5. CONCLUSIONS

In this paper we have presented the basic mechanisms used in CONLAN to define and extend languages. The mechanism is based not only on the definition of new types and operations but also on the extension of the syntax of a base language.

The concept of the interpreter is basic to the family of languages. The interpreter provides the basic counters used to model elapsed time (*t*@ and *s*@) and the detection of error conditions (error@). It can also be augmented in a controlled manner by the toolmakers. This is achieved by the definition of special functions and activities which can then be invoked automatically by the interpreter. In this paper we have shown a few of these operations (pack@, content@, finint@ and finstep@) which are used to provide restricted dereferencing and signal growth mechanisms.

A full coverage of the base language is outside the size limitations of this paper. Enough has been presented however, to motivate the reader to appreciate the power of the notation and its usefulness in the development of a comprehensive family of languages for describing the behavior and interconnection of hardware components.

6. ACKNOWLEDGMENTS

The authors are indebted to Bell Northern Research (Ottawa), Sperry Univac (Philadelphia), Office of Naval Research, Ballistic Missile Defense Advanced Technical Center (Huntsville), IRIA (Paris), Busdesministerium fur Forschung und Technologie (Bonn), Siemens (Munich), and Fujitsu (Tokyo) for their interest and support, Professor Yaohan Chu for his early contributions, and in particular to Professor Jack Lipovski for his help and unwavering confidence in the group.

7. REFERENCES

1. Piloty, R., Barbacci, M., Borrione, D., Dietmeyer, D., Hill, F., and Skelly, P., "CONLAN—A Formal Construction Method for Hardware Description Languages: Basic Principles," *Proceedings National Computer Conference*, Volume 49, Anaheim, California, 1980.
2. Piloty, R., Barbacci, M., Borrione, D., Dietmeyer, D., Hill, F., and Skelly, P., "CONLAN—A Formal Construction Method for Hardware Description Languages: Language Application," *Proceedings National Computer Conference*, Volume 49, Anaheim, California, 1980.